4
Estudos de Caso

Este capítulo detalha a utilização da ferramenta GenArch na derivação automática de dois estudos de caso diferentes: (i) framework JUnit; e (ii) uma linha de produtos de softwares para jogos de celular. Para cada um dos estudos de caso é apresentado: (i) a versão inicial dos modelos de derivação, criados automáticamente a partir dos elementos de implementação; (ii) a versão final dos respectivos modelos; e (iii) um exemplo de instanciação.

4.1.
Framework JUnit

Essa seção apresenta, detalhadamente, como a abordagem implementada pela ferramenta GenArch, descrita no capítulo anterior, foi aplicada ao framework JUnit e permite a derivação automática de instâncias desse framework. O JUnit foi escolhido por ser um framework de pequeno porte e amplamente conhecido. Acredita-se que estas características possam facilitar o entendimento do processo de adoção da abordagem.

O propósito principal do framework JUnit é permitir o projeto, implementação e execução de testes de unidade em aplicações Java. Este trabalho utiliza uma implementação refatorada do framework JUnit (Kulesza et al. 2007) usando a linguagem AspectJ (Laddad 2003). Programação orientada a aspectos foi utilizada nessa versão para prover uma melhor modularização de algumas características opcionais do JUnit. Duas funcionalidades do componente `extensions` foram implementadas como aspetos: (i) execução repetida de casos de teste; e (ii) execução concorrente de suítes de testes.

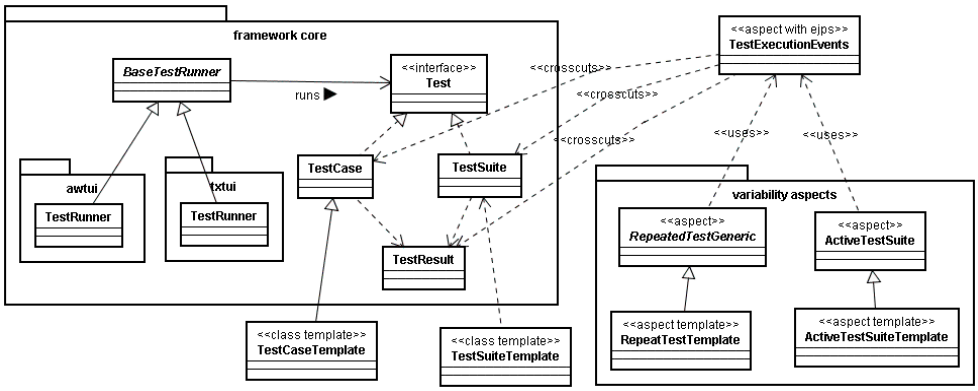


Figura 12. Arquitetura Orientada a Aspectos do JUnit

A Figura 12 apresenta os principais elementos da arquitetura do JUnit. Em particular, neste estudo de caso são considerados os seguintes componentes do framework:

- ***Core*** – contém as classes responsáveis por especificar o comportamento básico para execução de casos e suítes de testes. Os principais pontos de extensão (*hot-spots)* disponíveis neste componente são implementados pelas classes `TestCase` e `TestSuite`. Os usuários do *framework* precisam estender essas classes para criação de casos de testes específicos para suas aplicações;
- ***Runner*** – este componente é responsável por oferecer uma interface que facilite a execução e visualização do resultado de casos e suítes de testes. O JUnit fornece três implementações alternativas de interface: (i) baseada em linha de comando; (ii) baseada em componentes visuais da biblioteca Java AWT; e uma última (iii) baseada em componentes visuais da biblioteca Java Swing. Este estudo de caso implementa somente as duas últimas alternativas;
- ***Extensions*** – responsável por definir um conjunto de funcionalidades que estendem comportamentos básicos do framework JUnit. Exemplos de funcionalidades disponíveis são: (i) uma suíte de teste para executar várias outras suítes de teste em *threads* separadas; (ii) uma suíte de testes que permite a execução repetida de casos de teste; e (iii) um teste *decorator* que permite a execução separada de casos de teste. Este estudo de caso implementa somente as duas primeiras alternativas.

4.1.1.
Anotando código do JUnit com Características e Variabilidades

Seguindo os passos descritos na Seção 3.1, inicialmente, na preparação do JUnit para instanciação automática, foram criados um conjunto de anotações GenArch no código-fonte dos elementos de implementação. Como as anotações são geralmente inseridas no código de elementos de implementação que representam variabilidades da LPS ou pontos flexíveis de frameworks, as classes `TestCase` e `TestSuite`; e os aspectos `RepeatedTestGeneric` e `ActiveTestSuite` foram anotados.

A Figura 13 mostra a classe abstrata `TestCase` do framework JUnit marcada com duas anotações GenArch. A anotação `@Feature` indica que a classe `TestCase` implementa a características **Test Case** que, por sua vez, possui como pai a característica **Test Suite**. A anotação na Figura 13 também indica que a característica é obrigatória. Isso significa que todas as instâncias do framework JUnit requerem no mínimo uma implementação dessa classe. A anotação `@Variability` especifica que a classe `TestCase` é um ponto de extensão do framework JUnit. Essa anotação indica para a ferramenta GenArch que tal classe representa um ponto flexível, necessitando ser especializada no momento da criação de casos de testes (uma instância do JUnit) para uma aplicação Java. Nesse caso, um template responsável por gerar uma implementação padrão para o respectivo ponto de extensão é automaticamente criado pela ferramenta. A próxima subseção 4.1.2 mostra como essas anotações são processadas pela ferramenta GenArch para gerar a versão inicial dos modelos de derivação e dos templates.

```
@Feature(name="Test Case",parent="Test Suite",
                                  type=FeatureType.mandatory)
@Variability(type=VariabilityType.hotSpot,
                           feature="Test Case")
public abstract class TestCase extends Assert implements Test {
  private String fName;
  public TestCase() {
     fName= null;
  }
  public TestCase(String name) {
     fName= name;
  } ...
}
```

Figura 13. Classe `TestCase` anotada

4.1.2.
Gerando versões iniciais dos modelos

No segundo passo da abordagem, uma versão inicial de cada modelo GenArch é produzida automaticamente. A (a) exibe, por exemplo, a versão inicial do modelo de implementação do framework JUnit. Cada pacote foi convertido para um componente e cada elemento de implementação foi convertido para a sua respectiva abstração no modelo de implementação. Como foi mencionado na seção 3.1, o modelo de implementação é criado somente para fornecer uma representação visual dos elementos de implementação da LPS.

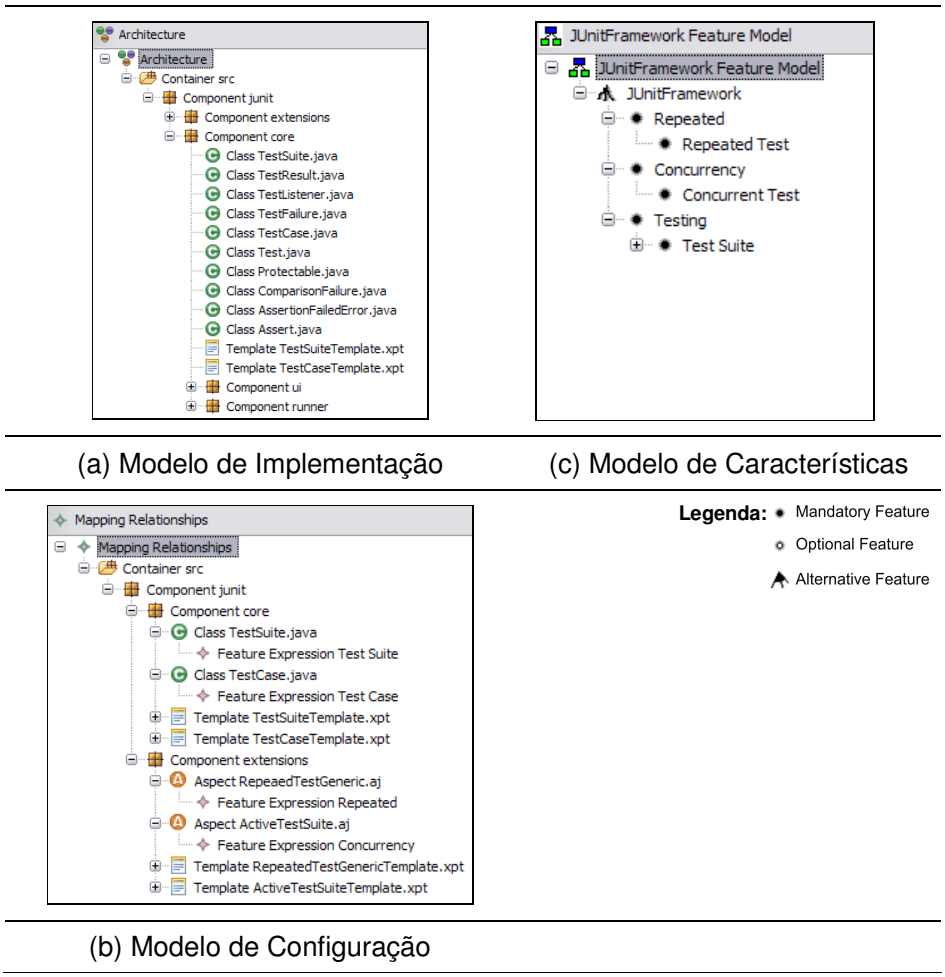


Figura 14. Versão Inicial dos Modelos GenArch do JUnit

A (c) mostra o modelo de característica parcial, gerado para o framework JUnit. Esse modelo contém, por exemplo, as características **Test Suite** e **Test**

Case, a última sendo criada a partir da anotação `@Feature` presente no elemento de implementação da Figura 13. A anotação `@Variability`, presente nesse mesmo elemento, demandou a criação do template `TestCaseTemplate`, como pode ser visto na (a). Este template será utilizado no momento da derivação do framework JUnit para gerar subclasses de `TestCase` para uma aplicação Java específica que será testada. Somente a estrutura do template, com a implementação de métodos concretos vazios ou pré-definições de pointcuts, é criada automaticamente. Essa criação é baseada na implementação do respectivo elemento (classe abstrata, interface ou aspecto abstrato) anotado. A próxima seção mostra como o código de um template pode ser customizado usando informações coletadas a partir das instâncias dos modelos de característica e implementação.

A (b) mostra o modelo de configuração inicial do framework JUnit, criado a partir do processamento das anotações. Como pode se observado, o aspecto `RepeatedTestAspect` depende explicitamente da característica **Repeated**. Esse mapeamento foi criado a partir da anotação `@Feature` marcada no aspecto `RepeatedTestGeneric`,

Figura 15.

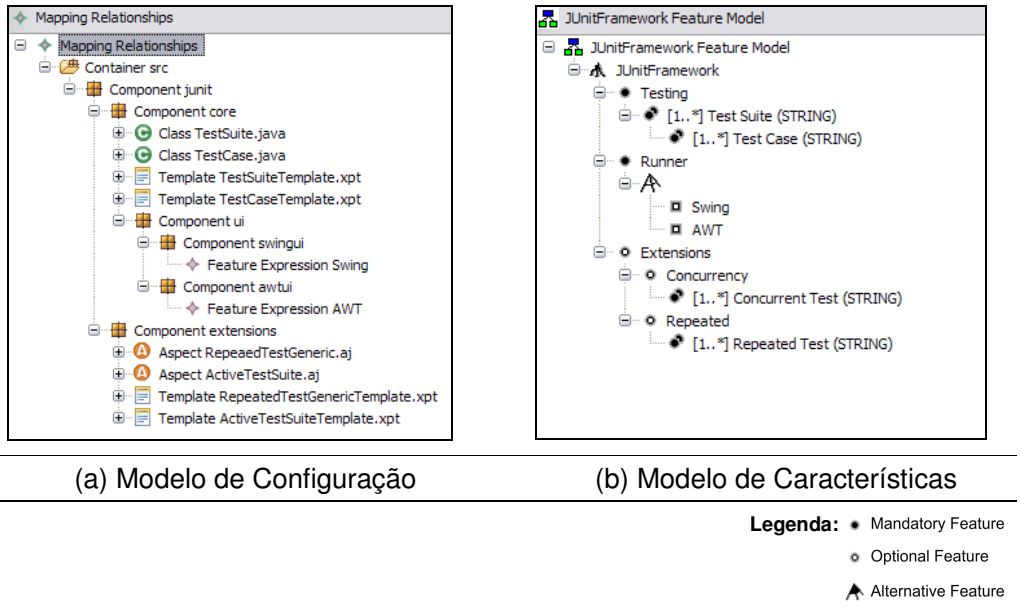
```
@Variability(feature="Repeated Test",type=VariabilityType.hotSpotAspect)
@Feature(name="Repeated Test", parent="Repeated",
                                        type=FeatureType.mandatory)
public abstract aspect RepeatedTestsGeneric {

       public abstract pointcut testExecution(TestResult result);

       public pointcut testCaseExecution (TestCase testCase,
                                              TestResult result):
              testExecution(result) &&
              target(testCase) && target(TestCase);

       public abstract int getTimesRepeat();

       void around(TestCase testCase, TestResult testResult):
              testCaseExecution(testCase, result) {
              for (int i=0; i < getTimesRepeat(); ++i){
                     proceed(test, testResult);
              }
       }
}
```

Figura 15. Aspect RepeatedTestGeneric anotado

Na abordagem, templates dependem necessariamente de uma ou mais características. A (b) mostra o template `RepeatedTestAspectTemplate`

dependendo explicitamente da característica **RepeatedTest**. Isso significa que, durante a derivação/instanciação do JUnit, este template só será processado se a característica **RepeatedTest** tiver sido selecionada na instância do modelo de característica em uso.

Após a geração das versões iniciais dos modelos, o engenheiro de domínio pode refiná-los incluindo, modificando ou removendo qualquer característica, elemento de implementação ou mapeamento. A Figura 16 mostra uma visão da versão final dos modelos do framework JUnit.



(a) Modelo de Configuração (b) Modelo de Características

Figura 16. Modelos GenArch do JUnit – Versão Final

No modelo de características, Figura 16(c), as características **Runner**, **AWT** e **Swing** foram adicionadas. Cada uma dessas características representa um tipo de interface de execução dos testes do JUnit. No modelo de configuração, Figura 16(b), os componentes que implementam a interface de execução dos testes não foram configurados inicialmente pela ferramenta e tiveram de ser configurados manualmente. Nesse caso, foram criados mapeamentos entre os componentes (pacotes Java) que implementam cada uma das interfaces de execução e suas respectivas características (TXT, AWT e Swing). Tais mapeamentos foram criados a partir da janela, mostrada na Figura 17, que permite após a seleção de um elemento no modelo de implementação, o mapeamento entre o elemento selecionado e uma expressão lógica de

características. A expressão pode ser criada a partir da seleção das características na lista superior ou a partir da especificação textual da expressão.

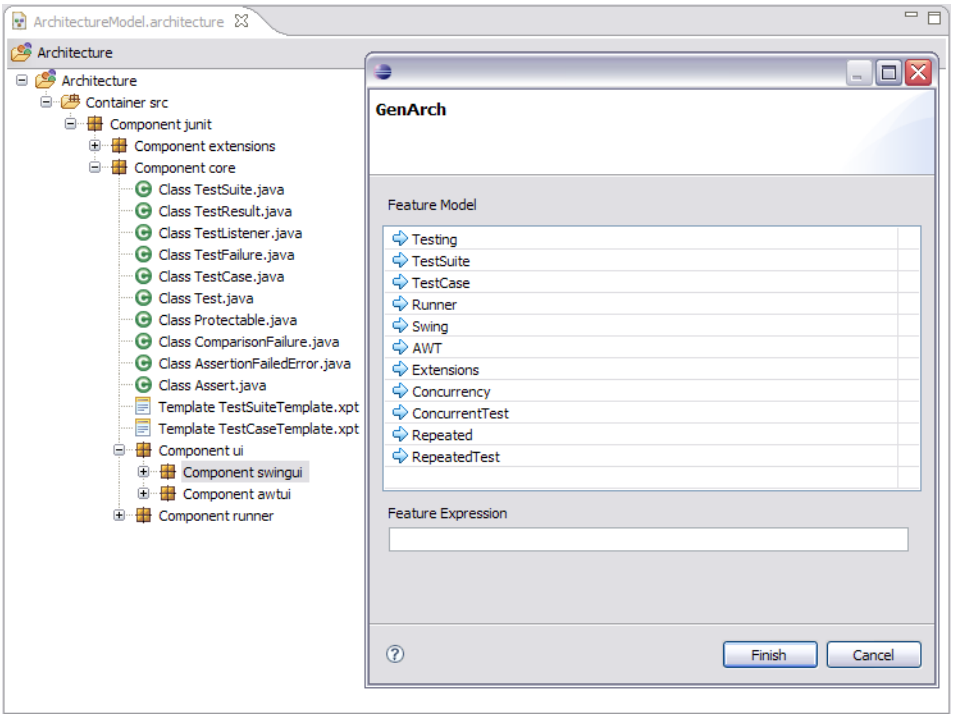


Figura 17. Mapeamento entre um elemento de implementação e uma característica

4.1.3. Implementando Variabilidades do JUnit com Templates

A Figura 18 mostra o código de template `TestSuite` implementado na linguagem XPand (openArchitectureWare 2008). Este template é utilizado para gerar suítes de teste específicas para uma aplicação Java. A definição `IMPORT` permite o uso de todos os tipos e extensões presentes no *namespace* indicado (linha 1). Essa definição é equivalente ao comando de importação da linguagem Java. Templates GenArch devem necessariamente importar o *namespace* `br::pucrio::inf::les::genarch::models::instance`, pois este contém as classes que representam os meta-modelos utilizados pelos templates. A definição `EXTENSION` (linha 2) permite estender templates XPand com novos métodos. Para terem acesso a um conjunto de classes que facilitam o acesso as informações disponibilizadas nos modelos de características e implementação, templates GenArch devem importar o *namespace* `br::pucrio::inf::les::genarch::models::Model`. A definição `DEFINE`

determina o corpo do template. Nessa construção deve se especificar o elemento que fornece as informações para o processamento do template. O template da Figura 18, por exemplo, está utilizando as informações presentes na classe `Instance`.

```
01 «IMPORT br::pucrio::inf::les::genarch::models::instance»
02 «EXTENSION br::pucrio::inf::les::genarch::models::Model»
03 «DEFINE Main FOR Instance»
04
05 «FOREACH configurations("testing/testSuite",featureElements) AS
06     testSuiteFeature»
07
08 «FILE testSuiteFeature.attribute + ".java"»
09 package junit.framework;
10 public class «testSuiteFeature.attribute» {
11     public static Test suite() {
12             TestSuite suite = new TestSuite();
13             «FOREACH testSuiteFeature.features AS child»
14             suite.addTestSuite(«child.attribute».class);
15             «ENDFOREACH»
16             return suite;
17     }
18 }
19 «ENDFILE»
20 «ENDFOREACH»
21 «ENDDEFINE»
```

Figura 18. TestCaseTemplate

Dentro da marcação `DEFINE` (linhas 3 a 21) é definida a seqüência de comandos responsáveis pela customização do template. A definição `FOREACH` permite a execução repetida de um bloco de comandos. No caso do `FOREACH` na linha 5, o bloco de comandos será executado uma vez para cada ocorrência da característica **Test Suite** na configuração do modelo de características. A marcação `FILE` (linha 8) especifica o arquivo que será escrito o resultado do processamento do template. A Figura 18 indica que o nome do arquivo de saída será o valor da propriedade `attribute`, da característica que esta sendo manipulada pelo template, acrescido da string ".java", que corresponde a extensão de uma classe Java. Os seguintes passos são realizados durante a execução do bloco de comando interno ao `FOREACH` (linhas 9-18): (i) o nome do caso de teste é obtido a partir da propriedade `attribute`; e (ii) os casos de teste para serem inseridos neste suíte de teste são obtidos através da propriedade `child.attribute` da feature `child`. A construção `FOREACH` permite o processamento das subcaracterísticas da característica **Test Suite** que está sendo processada.

4.1.4.
Instanciando o JUnit

O último passo do estudo de caso consiste na instanciação automática do framework JUnit. A derivação de um produto na ferramenta é organizada nos seguintes passos: (i) escolha e configuração das características que serão inclusas no produto; (ii) fornecimento da configuração do modelo de característica criado e do projeto Eclipse onde o produto será derivado para a ferramenta; e (iii) processamento de todos os modelos – a ferramenta utiliza os modelos para decidir quais elementos de implementação farão parte do produto final sendo derivado.

A Figura 19 exibe uma configuração do modelo de característica que especifica uma instância do framework JUnit. De acordo com a configuração do modelo de características da Figura 19, a ferramenta: (i) gera uma instância concreta do ponto de extensão **Test Suite** e três instâncias concretas do ponto de extensão **Test Case**; (ii) copia os arquivos correspondentes a implementação da interface com usuário do Swing; e (iii) instancia o JUnit com suporte a execução repetida de uma suíte de testes. Nesse caso, a suíte de testes **GenArchTestSuite**, especificada na propriedade `attribute` da características **Repeated Test**, será estendida com essa funcionalidade.

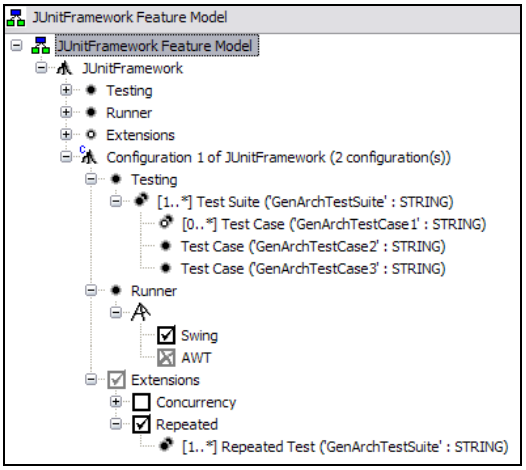


Figura 19. Configuração do modelo de característica

De acordo com a configuração da características do tipo **Test Suite** na Figura 19, e com o template `TestSuiteTemplate` (Figura 18), que é responsável por gerar instâncias concretas de suítes de teste, uma classe concreta, denominada `GenArchTestSuite`, será gerada. A customização do

template se dá a partir da propriedade `attribute` da característica **Test Suite**, como apresentado na Seção 3.2.3. A configuração da Figura 19, ainda especifica que três casos de teste (`GenArchTestCase1`, `GenArchTestCase2` e `GenArchTestCase3`), associados a características **Test Suite**, devem ser gerados. Esses casos de teste serão gerados pelo template `TestCaseTemplate`. Essas classes são agrupadas pela suíte de testes `GenArchTestSuite` gerada a partir da configuração da característica **Test Suite**, como descrito anteriormente na Seção 4.1.3 e mostrado na Figura 19. O número de vezes que cada um desses templates será executado é correspondente ao número de vezes que as características (**Test Suite** e **Test Case**) aparecem na instância do modelo de características.

```
01 «IMPORT br::pucrio::inf::les::genarch::models::feature»
02 «DEFINE Main FOR Feature»
03 «FOREACH configurations("extensions/repeated/repeatedTest",
04                                              featureElements)
05           AS repeatedTestFeature»
06 «FILE "Repeatedrepeated" + TestFeature.attribute + ".java"»
07 package junit.core;
08
09 public aspect Repeated«repeatedTestFeature.attribute» extends
10                                              RepeatedTestGeneric {
11
12     public pointcut testExecution(TestResult result):
13            call (void «repeatedTestFeature.attribute».run(TestResult))
14            && args(result);
15
16 }
17 «ENDFILE»
18 «ENDFOREACH»
19 «ENDDEFINE»
```

Figura 20. RepeatedTestGenericTemplate

A seleção da característica **Repeated** na configuração do modelo de características da Figura 19, informa a ferramenta que a instância do JUnit sendo criada deverá incluir a funcionalidade de execução repetida de suítes de teste. No caso da configuração da Figura 19, uma suíte de teste, **GenArchTestSuite**, vai ser executada repetidas vezes. Essa funcionalidade é contemplada pelo aspecto `RepeatedGenArchTestSuite` que é uma instância concreta do aspecto `RepeatedTest`. A geração do aspecto `RepeatedGenArchTestSuite` é feita a partir do template `RepeatedTestGenericTemplate`, da Figura 20. Durante a geração desse template, a ferramenta utiliza a informação contida na propriedade `attribute` da característica **Repeated Test** para gerar uma instância concreta do *pointcut* `testExecution`. Essa informação também é utilizada para customizar o nome

do aspecto gerado. No caso do template da Figura 20, o nome dos aspectos gerados é composto pela palavra Repeated seguida do nome da suíte de teste associada, definida na propriedade `attribute`. A definição `FOREACH` permite que o template seja executado para diferentes configurações da característica **Repeated Test**. Isso significa que diversos aspectos, cada um permitindo a repetição de diferentes suítes de teste, podem ser gerados.

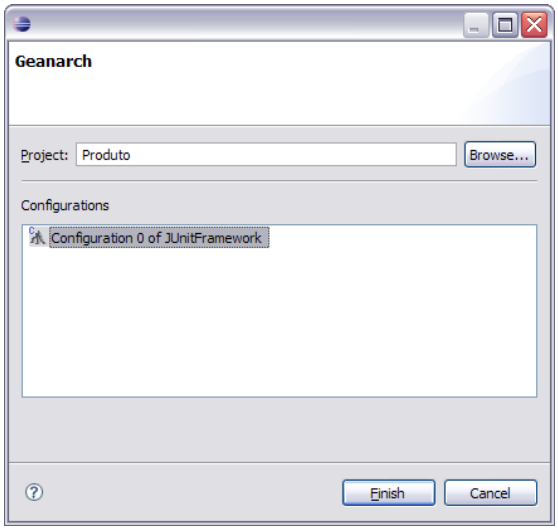


Figura 21. Janela de derivação

No final da derivação, todos os elementos que implementam o núcleo da LPS (que são os elementos obrigatórios), as classes e aspectos concretos gerados a partir dos templates e o aspecto abstrato `ConcurrentTestSuite` são copiados para um projeto Eclipse, especificado pelo engenheiro de aplicação, de acordo com a Figura 21, em seus respectivos pacotes. O código gerado deve ser finalmente completado de forma a implementar uma instância completa do framework JUnit.

4.2. Linha de Produtos para Jogos J2ME

4.2.1. Descrição do Estudo de Caso

Essa seção descreve a adoção da ferramenta GenArch na preparação e instanciação de uma LPS para jogos J2ME. Atualmente J2ME é uma das

principais tecnologias utilizadas no desenvolvimento de aplicação móvel, com os jogos sendo o principal tipo de aplicação desenvolvida nessa tecnologia. Muitos jogos J2ME podem ser vistos como LPS, principalmente porque necessitam ser adaptados e portados para diferentes dispositivos de forma a atingir diversos segmentos de mercado. É fato que esses dispositivos variam em termos de recursos disponíveis, tais como processador e memória. Dessa forma, é necessário fazer adaptações de acordo com as propriedades específicas de cada dispositivo. No entanto, as funcionalidades obrigatórias de cada jogo permanecem as mesmas, sendo essas definidas por um *game engine*. Um *game engine* define basicamente uma máquina de estados onde a transição de um estado para outro é definida pelo tempo que passa entre diferentes ações do usuário. Uma mudança de estado traz impacto sobre os objetos do jogo (tais como, atores e ambiente) e na forma como eles interagem. O redesenho de cada objeto do jogo ocorre tipicamente, após cada mudança de estado.

O jogo que envolve o nosso estudo de caso, denominado *Rain of Fire*, é um projeto comercial desenvolvido pela Meantime Móbile Creations[1]. Este jogo oferece as seguintes variabilidades: (i) imagens opcionais – a supressão de algumas imagens do jogo (exemplo: nuvens passando no fundo) não essenciais para sua execução, optimizam o desempenho do jogo em dispositivos com poucos recursos; (ii) APIs de desenho proprietárias – as imagens do jogo são desenhadas em diversos lugares e transformadas (rotacionadas, modificadas) em situações específicas, através de APIs proprietárias, referentes a cada dispositivo; (iii) carregamento das imagens – as imagens do jogo podem ser carregadas usando uma política de carregamento sob demanda, quando há mudanças na tela ou uma política de carregamento na inicialização, quando todas as imagens são carregadas de uma vez; (iv) a língua usada no jogo (Português, Inglês); (v) o modelo do celular onde o jogo será instalado; e (vi) mapeamento do teclado de cada dispositivo (Motorola ou Nokia).

Muitas das variabilidades do jogo *Rain of Fire* foram originalmente implementadas usando compilação condicional. Para melhorar a modularização e gerenciamento das variabilidades (Alves et al. 2005; Kulesza et al. 2006), o jogo foi refatorado e implementado usando programação orientada a aspectos (Filman et al. 2005). A implementação utilizando programação orientada a aspectos trouxe vários benefícios, como: (i) simplificação do núcleo do jogo com a extração de várias peças estáticas do código, o que é um resultado do uso da

[1] www.meantime.com.br

compilação condicional; e (ii) a capacidade de plugar/desplugar os aspectos da implementação do núcleo da LPS.

As próximas seções descrevem como a arquitetura da LPS do jogo *Rain of Fire* foi preparada para ser derivada automaticamente na ferramenta GenArch.

```
@Feature(name="On Demand",parent="Image Loading",
                        type=FeatureType.alternative)
public privileged aspect LoadImgOnDemand {
    after() : ResourcesEvents.loadingImages() {
          Resources.loadGameImages();
    }
    ...
}
```

```
@Feature(name="On Init",parent="ImageLoading",
                        type=FeatureType.alternative)
public privileged aspect LoadImgOnInit {
    after() : ResourcesEvents.loadingImages() {
          Resources.loadGameImages();
    }
    ...
}
```

Figura 22. Aspecto LoadImgOnDemand e LoadImgOnInit com anotações GenArch

4.2.2. Anotando Características na LPS para Jogos J2ME

O primeiro passo na preparação da LPS do jogo *Rain of Fire* para ser automaticamente derivada foi à anotação das respectivas variabilidades no código-fonte. Foram inseridas anotações `@Feature` em todos os aspectos e classes que modularizam as variabilidades do jogo. A exibe, por exemplo, o código dos aspectos `LoadImgOnDemand` e `LoadImgOnInit`, com a anotação `@Feature`. Essas anotações informam que esses aspectos representam implementações alternativas (**On Demand** e **On Init**) da característica **Image Loading**. A maioria dos aspectos foi anotada de forma similar, pois são implementações alternativas das variabilidades presentes no jogo *Rain of Fire*. Alguns aspectos também representam características opcionais, este é o caso, por exemplo, do aspecto `Cloud`, que implementa a imagem das nuvens passando no fundo do jogo.

4.2.3. Gerando os Modelos do jogo *Rain of Fire*

A exibe cada um dos modelos gerados. O modelo de característica contém todas as variabilidades anotadas e seus respectivos tipos ((c)). A

característica opcional **Cloud** representa as imagens opcionais que podem ser ou não incluídas no jogo. Várias características alternativas foram codificadas para endereçar as variações nos dispositivos onde o jogo será executado, como: **Cell Model**, **Flip**, **Key Mapping** e **Image Loading**. A característica **Image Loading**, por exemplo, define duas políticas alternativas (**On Demand** e **On Init**) baseadas na memória disponível em cada dispositivo.

A (a) ilustra o modelo de implementação do jogo *Rain of Fire*. Todas as classes/interfaces codificadas em Java e os aspectos codificados em AspectJ foram importados automaticamente a partir do código-fonte. O componente `rain` agrega todas as classes que implementam as características obrigatórias do jogo. Essas classes representam a *game engine*. Cada uma das variabilidades foi codificada através de um conjunto de aspectos organizados em componentes/pacotes separados. Exemplos desses componentes são: `bright`, `cell`, `crom`, `flip`, `keymapping` e `loadimg`. A única exceção é o componente `language` que foi implementado como classes (uma classe por linguagem). Nesse estudo de caso, não foi preciso à criação de templates, pois todas as variabilidades já estão modularizadas como aspecto ou classes alternativas e opcionais. Dessa forma não é necessária a customização de variabilidades durante o processo de derivação.

Finalmente, o modelo de configuração do jogo *Rain of Fire* também foi gerado automaticamente pela ferramenta. Tal modelo é exibido na (b). A figura exibe o mapeamento entre os elementos de implementação e as características do jogo *Rain of Fire*. Somente os elementos de implementação mapeados para expressões lógicas de características são exibidos nesse modelo. Os aspectos `LoadImgOnDemand` e `LoadImgOnInit`, por exemplo, que modularizam as políticas alternativas de carregamento de imagens, estão mapeados nas características **On Demand** e **On Init**, respectivamente. Isso significa que estes aspectos serão apenas derivados se suas respectivas características mapeadas estiverem selecionadas na instância do modelo de características em uso.

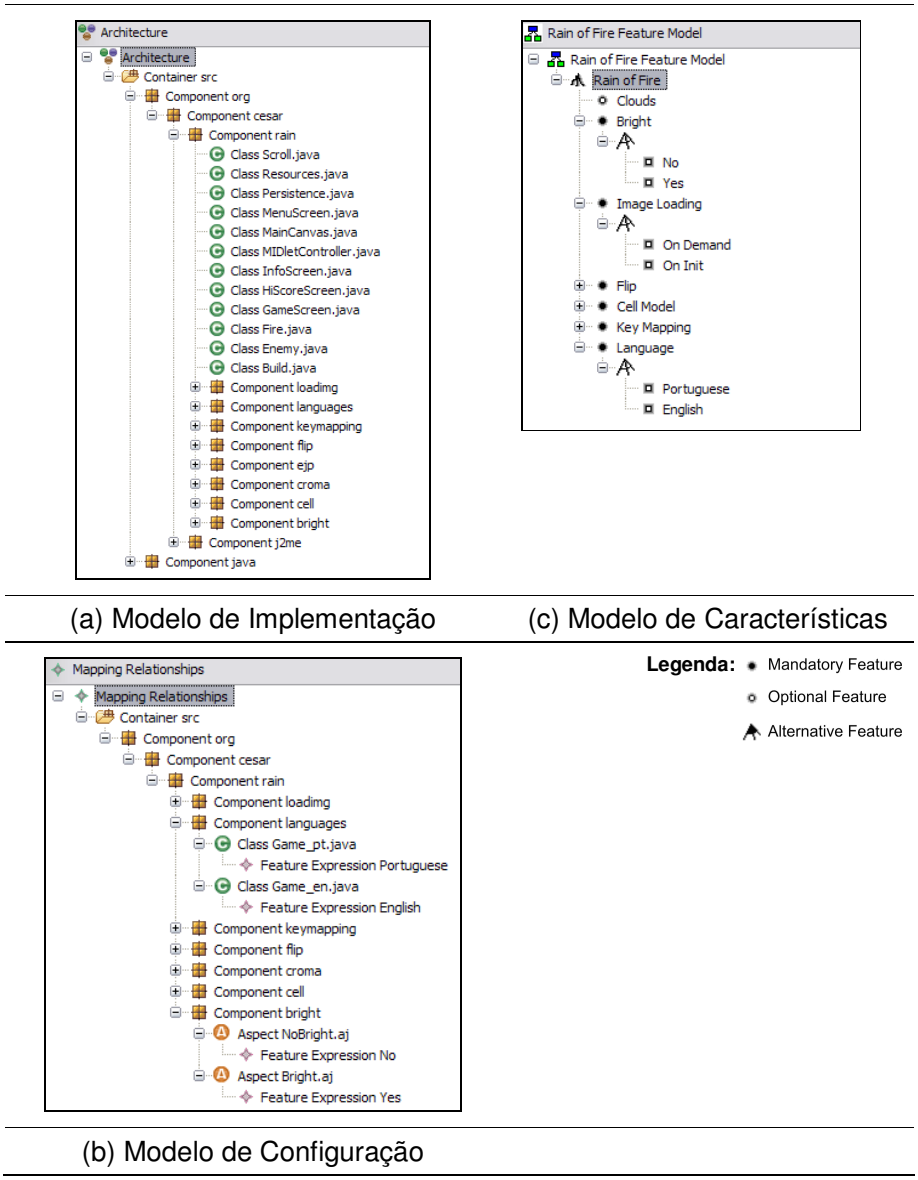


Figura 23. Modelos GenArch do jogo *Rain of Fire*

Nesse estudo de caso não foi necessário a criação de características, elementos de implementação ou mapeamentos adicionais nos modelos de características, implementação e configuração, respectivamente. Isso se deu porque todas as escolhas de implementação das variabilidades do jogo já estão codificadas e disponíveis, similar a um framework caixa-preta (*black box*) (Fayad et al. 1997). O simples uso das anotações GenArch foi suficiente para permitir a criação automática da versão final dos modelos de derivação. Essa característica destaca a eficiência da abordagem em lidar com LPS que possuem a maioria das suas variabilidades já implementadas.

4.2.4.
Derivando Jogos para Diferentes Modelos de Celular

Figura 24 mostra duas configurações do modelo de características que especificam diferentes versões do jogo *Rain of Fire* que foram derivadas. A configuração da

Figura 24(a) descreve uma versão do Jogo para um celular da marca Nokia (Nokia 2008) modelo S40. Esse modelo de celular é mais simples e possui menos recursos, dessa forma, as nuvens do fundo (característica **Clouds**) foram suprimidas, como pode ser visto na Figura 25(a). A política de carregamento das imagens sob demanda (característica **On Demand**) foi escolhida, por exigir menos recursos. A

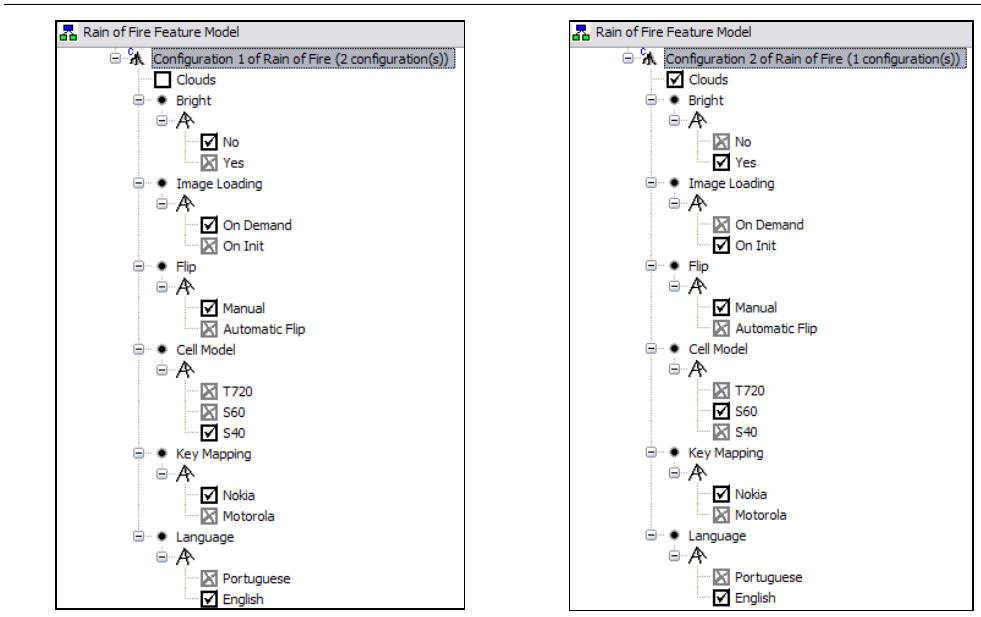


Figura 24. Configurações do modelo de característica do jogo *Rain of Fire*

Figura 24(b) descreve uma versão do jogo (Figura 25(b)) para o celular S60, também da marca Nokia. Esse modelo de celular possui mais recursos que o modelo apresentado anteriormente, o que possibilitou a seleção das nuvens (características **Clouds**) e permitiu o carregamento das imagens no início da execução do jogo (característica **On Init**). As outras características de cada uma das configurações foram selecionadas de acordo com as propriedades de cada modelo.

Figura 25. Jogo *Rain of Fire* executando em modelos diferentes de celulares

A derivação de cada um dos jogos a partir da LPS apresentada se deu, basicamente, através da cópia dos elementos de implementação, que estavam de acordo com a instância do modelo de características, para os seus respectivos pacotes em um projeto Eclipse novo.

4.3. Sumário

Esse capítulo apresentou em detalhes dois estudos de caso com o uso da ferramenta GenArch na preparação e derivação de LPS e frameworks. Primeiramente, a Seção 4.1 detalhou todos os passos que foram necessários para preparação do framework JUnit na ferramenta: anotação do código fonte; geração incial dos modelos de derivação; preparação da versão final dos modelos de derivação. Além disso, o processo de derivação de uma instância do JUnit também foi detalhado. A Seção 4.2 detalhou o processo de praparação de uma LPS para jogos J2ME e o processo de derivação de dois produtos diferentes.